ASSIGNMENT-6 REPORT

Abothula Suneetha (20CS10004)
Manami Mondal (20CS100)
Priyanshi Dixit (20CS10047)
Pranil Dey (20CS30038)

a. What is the structure of your internal page table? Why?

The page table used in our design is composed of two distinct parts. The first part is a classic page table, which is an array indexed by the local address that stores the physical address of the corresponding variable. This part is necessary to access the memory locations of the variables and modify or read their values based on the user's logical address. The second part is a list of variables that have been declared until now, again indexed by their local address. This list contains various details about the variables, such as their name, type, size, local address, array length, and a flag that is set to one when freeElem is called. This second part functions similarly to a symbol table and is necessary to retrieve information about the variables, such as type checking or freeing and allocating space. By indexing both parts by the local address, we achieve O(1) access to all information regarding the variable.

b. What are additional data structures/functions used in your library? Describe all with justifications.

ADDITIONAL FUNCTIONS

Stack *createStack() - This function is used to create a new stack data structure, which can be used to store a collection of variables. It returns a pointer to the newly created stack.

void push(Stack *s, Variable *x) - This function is used to push a new variable onto the stack. It takes in two parameters - a pointer to the stack and a pointer to the variable to be added. It adds the variable to the top of the stack.

void push(Stack *s) - This function is used to push a new element onto the stack. It takes in one parameter - a pointer to the stack. It adds an empty variable to the top of the stack.

Variable *top(Stack *s) - This function is used to retrieve the variable at the top of the stack without removing it. It takes in one parameter - a pointer to the stack. It returns a pointer to the variable at the top of the stack.

bool isEmpty(Stack *s) - This function is used to check if the stack is empty or not. It takes in one parameter - a pointer to the stack. It returns a boolean value - true if the stack is empty, and false otherwise.

ADDITIONAL DATA STRUCTURES

```
1.   typedef struct Node {
   int data;
   struct Node* next;
   struct Node* prev;
}Node;
```

The data structure is a doubly linked list node which consists of three parts:

1. An integer variable named data, which stores the actual data being stored in the node.
2. A pointer to the next node named next, which points to the next node in the list.
3. A pointer to the previous node named prev, which points to the previous node in the list.

Doubly linked lists are a type of data structure that allow for efficient traversal of a list in both forward and backward directions. They are commonly used in applications that require frequent insertion or deletion of nodes at arbitrary positions within the list.

```
2.   typedef struct BLOCK {
  char* name;
  int sz;
  int curr_sz;
  Node* list;
} BLOCK;
```

This code defines a struct named BLOCK which has the following members:

1. A pointer to a character array, name, that represents the name of the block.
2. An integer variable sz that represents the maximum size of the block.
3. An integer variable curr_sz that represents the current size of the block.
4. A pointer to a Node struct named list, which represents a doubly linked list.

This data structure is used to represent a block of memory that can be dynamically allocated and deallocated as needed. The name field can be used to identify the block, while the sz and curr_sz fields keep track of the maximum size and current size of the block, respectively. The list field is a pointer to a doubly linked list that stores the data elements within the block.

```
3.  typedef struct _Variable
{
  char *name;
  int type, size, localAddress, arrLen, isTobeCleaned;
} Variable;
```

The Variable structure represents a variable in a program. It has several fields including name, which is a character pointer representing the name of the variable; type, which is an integer representing the data type of the variable; size, which is an integer representing the size of the variable in memory; localAddress, which is an integer representing the local address of the variable in memory; arrLen, which is an integer representing the length of the array if the variable is an array and 1 otherwise; and isTobeCleaned, which is a flag indicating whether the variable is to be cleaned by the garbage collector or not. This structure is used to store information about variables in a program for various purposes, such as type checking and memory allocation.

```
4.   typedef struct _Stack
{
  Variable *stck[STACK_SIZE];
  int topIndex;
} Stack;
```

This is an implementation of a stack data structure using an array. The Stack structure has an array stck of Variable pointers with a fixed size of STACK_SIZE. The topIndex is an integer variable that represents the index of the topmost element in the stack.

This stack can store pointers to Variable structures, and is able to push and pop elements onto and off of the stack.

5. **typedef struct**

```
{
  int *pageTable[NUM_VARIABLES];
  int localAddress, maxMemIndex;
  int actualAddressToLocalAdress[MEM_SIZE / 4];
  Variable variableList[NUM_VARIABLES];
  Stack variableStack;
} Data;
```

This data structure, called Data, includes the following components:

- An array pageTable that is indexed by local address and stores the physical address of a variable.
- localAddress and maxMemIndex are integer variables that store the starting local address and the maximum index of memory used, respectively.
- An array actualAddressToLocalAddress that maps actual memory addresses to local addresses.
- An array variableList that contains details of all declared variables, such as their name, type, size, local address, array length, and a flag that indicates whether they should be cleaned by the garbage collector.
- A stack variableStack that stores the local addresses of declared variables.
- This implementation does not include any synchronization mechanisms or thread management data structures.

c. What is the impact of freeElem() for merge sort. Report the memory footprint and running time with and without freeElem (average over 100 runs).

The function freeElem() is important because it allows for the freeing of small linked lists that are created by merging. Without implementing this function, the memory usage would remain large because smaller lists are not deleted after they are used. Although it would take less time to empty the lists each time the capacity expires. For example, if there is a list with 50,000 elements, there would be 50,000 leaf nodes, and each element would take up 12 bytes. This means that the total size of leaf nodes alone would be 600KB. The actual memory usage is even higher because each element is allocated one page, so the actual memory space needed would be around 75MB, which is approximately 30% of the total memory.

The results of running a program for 100 iterations. In the first iteration, the average time taken was 2.52153 seconds, and the maximum page usage was 52729, which is ~30% of the memory. However, by calling the freeElem() function after each list is used, the memory footprint is reduced, although it takes more time to clear each list. In the second iteration, the average time taken was 2.62325 seconds, and the maximum page usage was only 385, which is ~0.22% of the memory.

d. In what type of code structure will this performance be maximized, where will it be minimized? Why?

The implementation of memory management would perform best when there is a small number of scope definitions in the code structure, but the performance would be negatively affected if there are multiple scope definitions, such as in recursive functions. This is because more time would be needed to maintain the scopes using stack frames and then delete the local lists when the scopes end.

e. Did you use locks in your library? Why or why not?

The design does not use locks since the processes or threads are not concurrently accessing the shared memory data structures .The design approach avoids race conditions by ensuring that multiple processes or threads do not try to access the shared memory data structure simultaneously. This can be achieved in different ways, depending on the design, including single access, independent copy, and non-concurrent access approaches.